# Una implementación completa del FQTrie

**Edgar Chávez**
Escuela de Ciencias Físico-Matemáticas
Universidad Michoacana
Morelia - México
elchavez@fismat.umich.mx

**N. Herrera, C. Ruano, A. Villegas**
Departamento de Informática
Univ. Nacional de San Luis
Argentina
nherrera,cmruano,anaville@unsl.edu.ar

**Resumen**

La próxima generación de manejadores de bases de datos deberá ser capaz de indexar datos multimedia y responder consultas de proximidad con tanta eficiencia como actualmente responden consultas de búsqueda exacta. Estas nuevas bases de datos se pueden modelar como un espacio métrico. Numerosas técnicas de indización han sido diseñadas para espacios métricos. Una de ellas es el *Trie de Consulta Fija (FQTrie)*, que ha demostrado experimentalmente tener un buen desempeño para resolver búsquedas por proximidad en espacios métricos.

En investigaciones anteriores hemos realizado trabajos en torno a mejorar la eficiencia del FQTrie desde dos tópicos diferentes: tiempo extra de CPU y tiempo de I/O. Con respecto al tiempo de CPU, hemos encontrado un método de discretización que logra mejorar la eficiencia del FQTrie. Con respecto al tiempo de I/O, hemos diseñado una técnica basada en el particionamiento del espacio que permite reducir el tiempo de I/O.

Nos proponemos lograr una implementación completa del FQTrie que funcione tanto en memoria principal como en memoria secundaria combinando las técnicas antes mencionadas.

## 1. Introducción

Los índices en memoria principal suponen que la base de datos completa y el índice caben en memoria principal. Esta suposición esta fundamentada por la gran cantidad de memoria RAM que poseen los ordenadores modernos, y se cumple en un gran número de casos; pero no en todas las aplicaciones. Para aplicaciones en donde la memoria principal no es suficiente para alojar el índice y/o los datos es necesario hacer consideraciones del número de accesos aleatorios que utilizará el algoritmo de indización. Los accesos en memoria secundaria pueden ser varios ordenes de magnitud más costosos que las consultas en memoria principal.

La problemática de búsquedas por similitud en bases de datos no tradicionales puede formalizarse por medio del modelo de *espacios métricos*. Un espacio métrico es un par $(\mathcal{X}, d)$, donde $\mathcal{X}$ es un conjunto de objetos y $d : \mathcal{X} \times \mathcal{X} \rightarrow R^+$ es una función de distancia que modela la similitud entre los elementos de $\mathcal{X}$. La función $d$ cumple con las propiedades características de una función

de distancia: $\forall x, y \in \mathcal{X},\ d(x,y) \geq 0$ (positividad), $\forall x, y \in \mathcal{X},\ d(x,y) = d(y,x)$ (simetría), $\forall x, y, z \in \mathcal{X},\ d(x,y) \leq d(x,z) + d(z,y)$(desigualdad triangular). La base de datos es un conjunto finito $\mathcal{U} \subseteq \mathcal{X}$.

Una de las consultas típicas que implica recuperar objetos similares de una base de datos es la *búsqueda por rango*, que denotaremos con $(q,r)_d$. Dado un elemento de consulta $q$, al que llamaremos *query* y un radio de tolerancia $r$, una búsqueda por rango consiste en recuperar aquellos objetos de la base de datos cuya distancia a $q$ no sea mayor que $r$.

Las búsquedas por similitud pueden ser resueltas trivialmente por medio de una búsqueda exhaustiva, con una complejidad $O(n)$. Para evitar esta situación, se preprocesa la base de datos por medio de un *algoritmo de indización* con el objetivo de construir una *estructura de datos o índice*, diseñada para ahorrar cálculos en el momento de resolver una búsqueda.

El tiempo total de resolución de una búsqueda puede ser calculado de la siguiente manera:

*T= #evaluaciones de $d\times$ complejidad(d) + tiempo extra de CPU + tiempo de I/O*

En muchas aplicaciones la evaluación de la función de distancia $d$ es muy costosa de calcular y por ello es importante reducir la cantidad de cálculos de distancia. Sin embargo, también es importante prestar especial atención al tiempo extra de CPU, dado que reducir este tiempo produce que en la práctica la búsqueda sea más rápida aún cuando estemos realizando la misma cantidad de evaluaciones de distancia. De igual manera, en aquellas aplicaciones donde el índice y los datos no pueden ser mantenidos en memoria principal, es importante reducir el tiempo de I/O.

Básicamente existen dos enfoques para el diseño de algoritmos de indización en espacios métricos: uno está basado en particiones compactas o tipo Voronoi y el otro está basado en pivotes [CNBYM01].

La familia de estructuras *FQ* (FQT [BYCMW94], FHQT [BYCMW94, BY97], FQA [CMN01], FQtrie [CF04]) forman parte del grupo de algoritmos basados en pivotes; cada una de ellas fue presentada como mejora a la anterior.

Hemos realizado trabajos en torno a mejorar la eficiencia del FQTrie. Estos trabajos se centran en la optimización de los dos últimos términos de la ecuación del tiempo total de una búsqueda (tiempo extra de CPU y tiempo de I/O) y mantienen una cantidad de cálculos de distancia sublineal respecto a la cantidad de elementos en la base de datos.

En [RCH04] se presenta un método de discretización, basado en los histogramas de distancias de los pivotes, que logra una implementación eficiente del FQTrie no sólo en términos de cantidad de evaluaciones de distancia de la función $d$, sino también en tiempo extra de CPU.

En [VCH04] se muestra una técnica que permite reducir el tiempo de I/O, la cual se basa en la idea de particionar la base de datos y agrupar en cada parte elementos similares.

Nos proponemos lograr una implementación totalmente eficiente del FQTrie, tanto en memoria principal como en memoria secundaria, combinando las técnicas antes mencionadas.

# 2. Fixed Queries Trie

El *Trie de Consulta Fija (**FQtrie** por sus siglas en Inglés)* fue presentado en [CF04] como una mejora al FQA [CMN01] utilizando tablas lookup para mejorar los tiempos de búsquedas. Este índice considera las firmas de los elementos como cadenas de caracteres de longitud fija, entonces al resolver una búsqueda se plantea el problema de encontrar igualdad entre cadenas. Desde este punto de vista podemos usar algunas de las estructuras específicamente diseñadas para reconocimiento de patrones. El FQtrie utiliza un *árbol Digital* o *Trie* para indizar las firmas de la base de datos.

El FQtrie representa cada firma haciendo uso de una función de discretización. Dicha función de discretización mapea números reales positivos en valores discretos de tamaño $b_p$ bits. Formalmente la

función de discretización se define como sigue:

$$\delta : \mathbb{R}^+ \times \mathbb{K} \rightarrow \{0, \ldots, 2^{b_p} - 1\}$$

De esta forma, para un elemento $o \in \mathbb{X}$ la firma de $o$ se define como la concatenación de la discretización de las distancias del objeto a cada pivote, en símbolos:

$$\delta^*(o) = \delta_{p_1}(d(o, p_1)) \bullet \delta_{p_2}(d(o, p_2)) \bullet \ldots \bullet \delta_{p_k}(d(o, p_k))$$

donde " $\bullet$ " es la operación de concatenación.

La principal ventaja de la función de discretización se centra en el grado de libertad en cuanto al uso de memoria disponible ya que podemos decidir cuantos bit asignar a cada pivote y determinar así el tamaño de la firma de cada objeto.

Además permite establecer un balance entre el poder de filtrado y el espacio utilizado ya que, a medida aumenta la cantidad de bit asignados a un pivote, aumenta la presición pero disminuye la cantidad de pivotes que pueden ser contenidos en la misma cantidad de espacio.

Buscar una cadena en un Arbol Digital toma tiempo proporcional a la cantidad de caracteres en ella, independientemente de la cantidad de elementos contenidos en el conjunto. Dada una cadena, los caracteres que la conforman son los que direccionan la búsqueda en el Arbol Digital. Para el caso del FQtrie, la búsqueda se realiza con la asistencia de la *tabla lookup* la cual contiene el conjunto de firmas de la búsqueda ($\delta^*(q, r)$). Esto implica una pequeña modificación en la forma de recorrer el árbol, permitiendo búsquedas de múltiples cadenas en el mismo recorrido. Esto es, estando en el nivel $i$, en lugar de seleccionar aquel nodo que concuerda con el $i$-ésimo caracter de la cadena, seleccionamos un nodo si alguna de las $i$-ésimas coordenadas en el conjunto de firmas de la búsqueda concuerda con el rótulo del nodo.

## 2.1. Diseño de Funciones de Discretización

El tipo de función de discretización y la *calidad* de los pivotes empleados son factores esenciales en la eficiencia del FQtrie. Esto se debe a que la función de discretización influye tanto en el espacio requerido por el índice (¿cuántos bits usamos por pivote?) como en el tiempo consumido por una búsqueda (¿qué tan bueno es el mapeo realizado por dicha función?). Para una cantidad $b_i$ de bits se pueden definir un número finito de reglas de discretización. Si bien la definición de la función de discretización no es esencial para la correctitud del método de filtrado sí lo es para su eficiencia.

Hemos diseñado funciones de discretización con el fin de lograr una implementación eficiente del FQtrie no sólo en términos de cantidad de evaluaciones de la función de distancia $d$, sino también en tiempo extra de *CPU*.

El diseño de estas técnicas se basó exclusivamente en lo que llamamos *histograma local de un pivote* $p$ [CNBYM01], el cual representa la distribución de distancias de $p$ a los elementos $u \in \mathbb{U}$. Este histograma permite visualizar la distribución de los elementos del espacio métrico respecto del punto $p$.

En [RCH04] presentamos varias alternativas de funciones de discretización así como los resultados de la evaluación experimental de las mismas sobre dos tipos de espacios métricos: diccionarios de palabras con función de distancia edición y documentos con función de distancia coseno.

En dicho artículo se presenta un método de discretización denominado $\delta_{\mu_p}$ el cual, utilizando sólo un bit por pivote (la cantidad más baja posible de memoria), alcanza una alta eficiencia al indizar el espacio de diccionarios. Este método permite mejorar notablemente los tiempos de búsquedas en el FQtrie, en condiciones de igualdad de memoria sin utilizar discretización.

No obstante, el comportamiento observado al indizar un espacio de documentos no fue muy alentador.

## 2.2. Método de Paginación para el FQTrie

Para reducir el tiempo que insume una búsqueda, es importante considerar si el índice y los datos pueden ser mantenidos en memoria principal, o si es necesario utilizar memoria secundaria para los índices y/o datos. En el primer caso el objetivo central es reducir los cálculos de distancia realizados. Pero, para los algoritmos en memoria secundaria, además de realizar pocos cálculos de distancia, se requiere que se realicen pocos accesos a disco.

Hemos trabajado para lograr un manejo eficiente de espacios métricos en disco, es decir, un manejo eficiente de espacios que excedan la capacidad de memoria principal. Para ello, se ha diseñado una técnica que particiona el espacio de manera tal que cada una de las partes entre en memoria principal. En cada parte se agrupan elementos similares y cada una de ellas se indexa en forma separada. Luego una consulta en el espacio métrico particionado se resuelve buscando en cada parte, lo que puede ser hecho en memoria principal. Esto implicaría un número constante de visitas a disco (la cantidad de partes generadas más la cantidad de páginas que contengan índices). Sin embargo, al agrupar elementos similares en cada parte, se evitan visitas a ciertas páginas de disco, ya que buscar en el índice de una parte puede indicar que en esa parte no hay elementos relevantes a la búsqueda, y no es necesario cargarla. Es aquí donde se evita un acceso a disco. La técnica de particionamiento diseñada se basa en la distancia LCS que calcula la longitud de la máxima subsequencia común entre dos cadenas. Esta técnica detecta grupos de elementos parecidos, denominados *t_aceptables*, a partir de los cuales genera la partición.

Se estudió el comportamiento de esta técnica sobre diccionarios de palabras con la función de distancia de edición y sobre documentos de texto con la función de distancia coseno [VCH04]. La técnica de particionado LCS demostró ser competitiva cuando utilizamos como espacio métrico diccionarios de palabras, logrando disminuir la cantidad de accesos a disco en un $21\,\%$ respecto de un particionado totalmente aleatorio. También demostró ser eficiente disminuyendo cantidad de accesos a disco requeridos ante una consulta respecto de no particionar la base de datos y dejar el manejo al sistema operativo. El comportamiento ha sido diferente cuando el espacio métrico utilizado son documentos de texto y los resultados no han sido tan alentadores respecto de no particionar la base de datos. En [VCH04] se concluye que, si los elementos de la base de datos son mayores que el tamaño de un página de disco, no es bueno particionar el espacio métrico ya que eso implica almacenar punteros en cada parte y los resultados han mostrado que, para estos casos, no es posible obtener ventaja de dividir el espacio métrico en partes, aún cuando la división se realiza en forma controlada.

# 3. Trabajo Futuro

Hemos encontrado un método que utiliza un bit por pivote (la mínima cantidad de memoria) y que tiene una eficiencia muy alta, y una técnica de particionamiento que permite disminuir la cantidad de accesos a disco en espacios que exceden la capacidad de la memoria principal. Intuimos que, al combinar ambas soluciones, podríamos lograr una implementación totalmente eficiente del FQtrie logrando reducir tanto el tiempo de CPU como el tiempo de I/O.

Sabiendo que para cada parte se mantiene sólo el conjunto de firmas y no un índice en particular particionamos el espacio con la técnica de partición implementada, creando las firmas de los elementos en cada parte utilizando $\delta_{\mu_p}$ como función de discretización. Con esto podremos hacer un manejo eficiente del FQTrie en memoria secundaria y a la vez obtener las ventajas provistas por la función de discretización cada vez que un índice requiera ser consultado en memoria principal.

Por otro lado, la función de discretización $\delta_{\mu_p}$ utiliza solo un bit de memoria para representar la distancia a cada pivote, generando firmas de tamaño mínimo. Esto favorece al manejo en memoria secundaria ya que al generar índices más pequeños se puede aumentar la cantidad de índices contenidos

en cada página, con lo cual se disminuye la cantidad de accesos a discos necesarios para recuperar los índices de todas las partes.

Otro punto en el que estamos trabajando es plantear nuevas funciones de discretización y nuevas técnicas de particionado del espacio métrico para aquellos espacios donde los resultados obtenidos no fueron favorables.

En cuanto a las funciones de discretización, estamos analizando nuevos diseños que no dependan de la media del histograma ya que en algunos casos (como lo es el espacio de documentos), la mayor concentración de elementos se identifica en más de un punto del histograma. En [BYBC$^+$03] se presenta el concepto de nucleo duro y nucleo blando. El nucleo duro del espacio está formado por los elementos que se encuentran en la zona central de los histogramas locales de varios pivotes, representando de esta forma una zona de búsqueda *d*ifícil de tratar. El nucleo blando del espacio está formado por el resto de los elementos no incluidos en el nucleo duro. En forma general la idea planteada es discretizar las distancias intentando diferenciar los elementos que pertenecen al nucleo duro del espacio.

En cuanto a técnicas de partición estamos analizando nuevas técnicas de particionado para el manejo eficiente en memoria secundaria de aquellos espacios métricos cuyos objetos son mayores que una página de disco.

# Referencias


[BY97] R. Baeza-Yates. Searching: an algorithmic tour. In A. Kent and J. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker Inc., 1997.

[BYBC$^+$03] R. Baeza-Yates, B. Bustos, E. Chávez, N. Herrera, and G. Navarro. *Clustering in Metric Spaces and Its Application to Information Retrieval*. Kluwer Academic Publishers, 2003. ISBN 1-4020-7682-7.

[BYCMW94] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. 5th Combinatorial Pattern Matching (CPM'94)*, LNCS 807, pages 198–212, 1994.

[CF04] E. Chávez and K. Figueroa. Faster proximity searching in metric data. In *Proceedings of MICAI 2004. LNCS 2972, Springer*, Cd. de México, México, 2004.

[CMN01] E. Chávez, J. Marroquín, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.

[CNBYM01] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, September 2001.

[RCH04] C. Ruano, E. Chávez, and N. Herrera. Discretización binaria del fqtrie. In *Actas del X Congreso Argentino de Ciencias de la Computación (CACIC'04)*, pages 100–111, Buenos Aires, Argentina, 2004.

[VCH04] A. Villegas, E. Chávez, and N. Herrera. Métodos de paginación para Índices métricos basados en pivotes. In *Actas del X Congreso Argentino de Ciencias de la Computación (CACIC'04)*, pages 306–316, Buenos Aires, Argentina, 2004.